

Министерство образования и науки Российской Федерации  
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий  
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра

# Удаление примитивных типов из высокоуровневого языка программирования для улучшения качества семантического анализа

**Автор:**

Студентка группы Б01-109а  
Ермолаева Варвара Дмитриевна

**Научный руководитель:**

Солдатов Антон Анатольевич

**Научный консультант:**

Солдатов Антон Анатольевич



Москва 2025

## Удаление примитивных типов из высокоуровневого языка программирования для улучшения качества семантического анализа

Ермолаева Варвара Дмитриевна

В современных управляемых языках программирования (таких как Java, C#, Kotlin) сохраняется исторически сложившееся разделение типов на примитивные (например, `int`, `boolean`) и объектные (ссылочные). Это разделение, хотя и направлено на повышение производительности, приводит к:

- Несогласованности в системе типов
- Усложнению семантического анализа
- Ограничению возможностей объектно-ориентированного программирования (например, отсутствие полиморфизма для примитивов, необходимость упаковки/распаковки)

Целью данной работы является улучшение качества семантического анализа за счёт устранения дуализма примитивных и объектных типов в высокоуровневом языке программирования с управляемой средой исполнения. Для этого предложена стратегия «Примитивы как оптимизация компилятора», при которой:

- Все типы унифицируются в рамках объектной модели
- Примитивные операции выполняются непосредственно над объектами (например, `Int` вместо `int`)
- Низкоуровневые оптимизации (замена объектов на примитивы) выполняются на этапе компиляции, не влияя на семантику языка

В работе проведён:

- Анализ существующих подходов (Java, C#, Kotlin)
- Разработана формальная спецификация унифицированной системы типов
- Реализованы:
  - Модификации семантического анализа для поддержки объектных примитивоподобных типов
  - Алгоритмы свёртки констант и преобразования типов
  - Механизм оптимизации AST для замены объектов на примитивы

Результаты показали, что предложенное решение:

- Упрощает семантический анализ за счёт устранения специальных правил для примитивов
- Сохраняет производительность благодаря оптимизациям на этапе компиляции
- Повышает удобство разработки, обеспечивая единообразие типов

**Ключевые слова:** система типов, примитивные типы, семантический анализ, оптимизация компилятора, унификация типов.

## Содержание

<b>1</b>	<b>Введение</b>	<b>5</b>
1.1	Типовая система . . . . .	5
1.2	Статическая и динамическая типизации . . . . .	6
1.3	Примитивные типы . . . . .	7
1.3.1	Роль в конструировании типов . . . . .	7
1.3.2	Основные характеристики . . . . .	7
1.3.3	Примеры в высокоуровневых языках . . . . .	7
1.4	Роль примитивных типов в управляемых средах исполнения . . . . .	8
1.4.1	Основные функции и компоненты среды исполнения . . . . .	8
1.4.2	Примитивные типы в управляемых средах исполнения . . . . .	9
1.5	Несогласованность примитивных типов в управляемых языках програм- мирования . . . . .	10
<b>2</b>	<b>Постановка задачи</b>	<b>13</b>
2.1	Контекст . . . . .	13
2.2	Задачи исследования . . . . .	13
2.3	Требования к решению . . . . .	14
2.3.1	Функциональные требования . . . . .	14
2.3.2	Нефункциональные требования . . . . .	15
2.4	Ожидаемые результаты . . . . .	16
<b>3</b>	<b>Обзор существующих решений</b>	<b>17</b>
3.1	Явное разделение с автоматическим преобразованием . . . . .	17
3.2	Единая иерархия типов с значимыми типами . . . . .	18
3.3	Примитивы как оптимизация компилятора . . . . .	19
3.4	Полное удаление примитивных типов . . . . .	20
<b>4</b>	<b>Исследование и построение решения задачи</b>	<b>22</b>
4.1	Архитектурное решение . . . . .	22
4.2	Обоснование выбора стратегии . . . . .	22
4.3	Описание решения . . . . .	23
4.4	Изменения в семантике языка . . . . .	23
4.4.1	Эквивалентные сценарии . . . . .	23
4.4.2	Новые допустимые конструкции . . . . .	24
4.5	План реализации . . . . .	24
<b>5</b>	<b>Описание практической части</b>	<b>25</b>
5.1	Арифметические операции . . . . .	25
5.2	Преобразования типов . . . . .	25
5.2.1	Основные функции . . . . .	25
5.3	Свёртка констант . . . . .	27
5.3.1	Основной алгоритм . . . . .	27
5.3.2	Поддерживаемые типы и операции . . . . .	28
5.3.3	Детали реализации . . . . .	28
5.3.4	Основные функции . . . . .	29
5.4	Оптимизация . . . . .	31
5.4.1	Класс UnboxVisitor . . . . .	31
5.4.2	Обрабатываемые узлы AST . . . . .	31
5.4.3	Механизм работы Visitor'ов . . . . .	31

5.4.4	Пример обработки CallExpression . . . . .	32
5.4.5	Логика преобразования типов . . . . .	32
5.4.6	Пример преобразования . . . . .	33
5.5	Тестирование . . . . .	33
5.6	Замеры производительности . . . . .	33
<b>6</b>	<b>Результаты и выводы</b>	<b>34</b>
6.1	Теоретические достижения . . . . .	34
6.2	Оценка эффективности . . . . .	34
6.3	Перспективы развития . . . . .	34
6.4	Практическая значимость . . . . .	35
	<b>Список литературы</b>	<b>36</b>

# 1 Введение

Типовая система языка программирования служит фундаментом для формального определения его спецификации, обеспечивает статический анализ программ, поддержку оптимизаций кода и способствует улучшению его структурированности и читаемости. Высокоуровневые языки программирования (Java, C#, Kotlin) предоставляют разработчику мощные абстракции и безопасность кода посредством автоматического управления памятью и объектно-ориентированных парадигм. Многие из этих языков сохранили в себе исторически сложившееся разделение типов на "примитивные" типы и "объектные" или "ссылочные", призванное потенциально улучшить производительность в некоторых случаях. Тем не менее, в то же время оно приводит к несогласованности в системе типов и усложняет достижение полностью единообразной и интуитивной модели программирования. Примитивные типы часто лишены полезных свойств объектов, таких как наследование, полиморфизм, вызов методов без явного упаковывания/распаковывания и использование в качестве аргумента для шаблонных типов.

В данной работе будут теоретически рассмотрены последствия устранения явной разницы между примитивными и объектными типами из высокоуровневого управляемого языка программирования и описаны этапы и результаты реализации.

## 1.1 Типовая система

Типовая система — следуя определению Пирса[1], это формальная синтаксическая и семантическая структура, сопоставляющая типы и программные конструкции (выражения, переменные, функции) для обеспечения чётко определённого вычислительного поведения. Её основные теоретические и практические назначения:

### 1) Корректность поведения программы

С введением типовой системы становится возможной статическая верификация соблюдения ограничений типов, не допускающая к выполнению неверно типизированные программы. Данный механизм гарантирует:

- *Безопасность выполнения* — предотвращение неопределённого поведения за счёт блокировки запрещённых операций (например, некорректные обращения к памяти в управляемых средах типа JVM (Java Virtual Machine)/CLR (Common Language Runtime))
- *Теоретическую обоснованность* — соответствие двум фундаментальным принципам типобезопасности по Райту-Феллейзену [2]:
  1. **Прогресс:** корректно типизированная программа не может застрять в промежуточном состоянии
  2. **Сохранение:** типы остаются согласованными на всех этапах вычисления

### 2) Уровень абстракции кода

Типовая система обеспечивает инкапсуляцию доменных инвариантов с помощью абстракций типов (абстрактные типы данных и интерфейсы) и позволяет формально специфицировать границы модулей.

### 3) Возможность оптимизаций

Наличие статической типизации позволяет применять оптимизации на этапе компиляции (например, специализация шаблонных типов и статическое разрешение методов) и минимизировать время исполнения в управляемых средах за счёт уменьшения количества динамических проверок типов.

## 1.2 Статическая и динамическая типизации

### 1. Статическая типизация (Java, C#, Kotlin):

- Валидность типа доказывается на этапе компиляции с помощью формальных суждений о принадлежности выражения к типу в данном контексте ( $\Gamma \vdash e : \tau$ )
- Гарантирует типобезопасность по Райту-Феллейзену
- Критично для управляемых языков программирования из-за возможности проверок безопасности памяти (например, верификатор байт-кода JVM) и эффективности Just-In-Time компиляции за счёт встраивания методов, ориентированных на конкретный тип

### 2. Динамическая типизация (Python, JavaScript):

- Типы функций и выражений определяются во время исполнения
- Нет формальной гарантии типобезопасности, но есть возможность менять поведение объектов, модулей и классов во время исполнения

Современные управляемые языки высокого уровня (Java, C#) преимущественно основаны на статической типизации, расширенной возможностями динамического анализа во время выполнения. Данный компромиссный подход позволяет сочетать строгость формальной верификации с практической гибкостью разработки, но противоречие между статическими гарантиями и динамической адаптируемостью продолжает оставаться предметом активных исследований.

Исторически и архитектурно, многие языки разделяют типы на две категории: **Примитивные типы** (далее – примитивные типы или примитивы) Представляют собой базовые значения (например, целые числа, числа с плавающей точкой, булевы значения, символы). **Ссылочные типы** (далее - ссылочные типы, объектные типы) Представляют собой экземпляры классов, которые хранятся в куче (heap) и управляются сборщиком мусора. Они наследуются от базового класса и обладают методами, полями и другими объектно-ориентированными свойствами.

## 1.3 Примитивные типы

**Примитивные типы** являются наиболее фундаментальным типом данных в языках программирования. В отличие от объектных типов, они:

- **Не ссылают на память**, а непосредственно являются значениями. У них отсутствуют:
  - заголовок объекта (object header)
  - таблица виртуальных методов (vtable)
  - ссылочная идентичность (identity)
- Имеют **фиксированную семантику**, определяющуюся стандартами языка (например, Java JLS §4.2 [3], C# ECMA-334 §8.3 [4])
- **Не могут быть подтипами** или наследоваться и являются атомарными (базовыми) элементами в иерархии типов

### 1.3.1 Роль в конструировании типов

В теории типов примитивы являются простейшими случаями конструирования типа и не разлагаются на более простые типы. Конструирование сложных типов происходит на основе:

- **Произведения** (кортежи, записи) – (Int, Bool)
- **Суммы** (типы-суммы, перечисления) – `data Maybe a = Nothing | Just a`
- **Функции** – `Int → Bool`
- **Рекурсивные типы** – `data List a = Nil | Cons a (List a)`

### 1.3.2 Основные характеристики

- **Представление в памяти**

Аллокация на стеке (за исключением глобальных переменных)

- **Производительность на уровне процессора**

1. Арифметические операции с примитивами компилируются непосредственно в нативные инструкции:

```
1      ; x86 assembly for 'a + b'
2      mov eax, [a]
3      add eax, [b]
```

2. **Локализация в кэше**

Линейная модель доступа к памяти (критично для больших вычислений)

### 1.3.3 Примеры в высокоуровневых языках

Тип	Java	C#
Целые числа	int, long	int, long
Числа с плавающей	float, double	float, double
Булевы значения	boolean	bool
Символ	char	char

## 1.4 Роль примитивных типов в управляемых средах исполнения

**Управляемая среда исполнения (Managed Execution Environment)** — это программный компонент или слой, который обеспечивает выполнение программного кода в контролируемой, безопасной и абстрагированной от низкоуровневых деталей среде. Она действует как посредник между скомпилированным (или интерпретируемым) кодом приложения и базовой операционной системой или аппаратным обеспечением.

### 1.4.1 Основные функции и компоненты среды исполнения

**Виртуальная Машина (ВМ) / Интерпретатор:** Ядро среды исполнения, которое выполняет промежуточный код (байт-код или Common Intermediate Language – CIL). Примеры включают Java Virtual Machine и Common Language Runtime для .NET. ВМ изолирует исполняемый код от конкретной аппаратной архитектуры и операционной системы.

**JIT-компиляция (Just-In-Time Compilation):** Для повышения производительности многие ВМ используют JIT-компиляторы. Они динамически переводят промежуточный код в машинный код непосредственно перед его выполнением. JIT-компиляторы могут применять оптимизации на основе профилирования времени выполнения, что часто позволяет достичь производительности, сопоставимой с нативно скомпилированным кодом.

**Загрузчики Классов/Сборок:** Эти компоненты отвечают за динамическую загрузку программных модулей (классов, библиотек, сборок) по мере их необходимости во время выполнения программы. Они также управляют разрешением зависимостей и проверкой целостности загружаемых компонентов.

**Система Управления Потоками:** Среда исполнения предоставляет интерфейс и механизмы для создания, управления и синхронизации потоков выполнения, обеспечивая эффективную поддержку параллельных вычислений.

**Система Безопасности:** Высокоуровневые языки программирования часто включают встроенные модели безопасности (например, песочницы), которые контролируют доступ исполняемого кода к системным ресурсам, таким как файловая система, сеть или другие процессы, что особенно важно для кода, загружаемого из недоверенных источников.



**Встроенные Сервисы:** К ним относятся механизмы обработки исключений, рефлексия (возможность интроспекции и модификации структуры кода во время выполнения), сериализация, а также поддержка взаимодействия с нативным кодом (Native Interface).

### 1.4.2 Примитивные типы в управляемых средах исполнения

Примитивные типы данных в контексте управляемых сред исполнения (таких как .NET CLR и Java VM) обладают характеристиками и принципами работы, отличающимися от неуправляемых сред, например, реализованных на языках C/C++.

#### Абстракция и безопасность

Управляемые среды исполнения предоставляют уровень абстракции, скрывающий детали низкоуровневого представления примитивных типов данных в памяти. Это обеспечивает платформонезависимость и переносимость программного обеспечения. Кроме того, такие среды реализуют строгую проверку типов как на этапе компиляции, так и во время исполнения, что служит профилактикой ошибочного использования типов данных и возникновения уязвимостей, связанных с выделением памяти и выходом за границы допустимых областей.

#### Гарантированная инициализация

В отличие от неуправляемых сред, где переменные могут содержать неинициализированные значения, управляемые среды гарантируют автоматическую инициализацию переменных примитивных типов значениями по умолчанию. Это исключает возможность непредсказуемого поведения, связанного с использованием мусорных значений памяти.

#### Сборка мусора

В управляемых средах примитивные типы данных, как правило, не подлежат сборке мусора напрямую. Обычно они размещаются в стеке либо, если выступают в роли компонентов объектов, внутри кучи. Освобождение памяти, занятой объектами (соответственно, и их полями-примитивами), происходит посредством встроенного механизма сборки мусора, что снимает с разработчика необходимость вручную управлять памятью.

#### Стандартное поведение

Управляемые среды исполнения гарантируют предсказуемое и стандартизированное поведение всех операций с примитивными типами данных, в том числе арифметических. Данная особенность повышает надежность и упрощает переносимость программ между различными аппаратными и программными платформами.

## Механизм упаковки и распаковки

Управляемые среды поддерживают механизм упаковки (*от англ. boxing, далее - боксинг, упаковка*) и распаковки (*от англ. unboxing, далее - распаковка, анбоксинг*), позволяющий преобразовывать значения примитивных типов в объекты и обратно. Это особенно актуально при взаимодействии с универсальными коллекциями и другими структурами, требующими объектного представления данных. Например, в среде C# значение типа `int` может быть преобразовано в объект (*boxing*) и обратно (*unboxing*).

## Примеры реализации

- **.NET (например, на C#):** Примитивные типы (такие как `System.Int32`) реализованы в виде структур и обеспечивают предсказуемое поведение за счёт встроенных механизмов среды.
- **Java:** Примитивные типы (`int`, `boolean`, `char`) имеют фиксированные размеры и поведение. Для поддержки упаковки и распаковки используются соответствующие классы-обёртки (`Integer`, `Boolean`, `Character`).

## 1.5 Несогласованность примитивных типов в управляемых языках программирования

В большинстве управляемых языков программирования (таких как Java, C#, Python) существует проблема несогласованности обработки примитивных типов данных и объектных типов в рамках единой системы типов. Эта асимметрия оказывает существенное влияние на архитектуру языка, его производительность и способность к применению принципов объектно-ориентированного программирования.

## Отсутствие единообразия в системе типов

Одной из ключевых проблем является отсутствие унифицированного представления примитивных и объектных типов в системе типов управляемого языка. Это бинарное разделение приводит к концептуальной и синтаксической асимметрии. Например, примитивные типы не могут быть `null` (без дополнительных обёрток), не поддерживают полиморфизм и не могут использоваться там, где требуется экземпляр класса `Object` (например, в универсальных коллекциях) без явного или неявного преобразования.

## Ограничения в объектно-ориентированных возможностях

Несогласованность примитивных типов создает существенные ограничения для применения объектно-ориентированных парадигм: инкапсуляция, наследование и полиморфизм.

- **Отсутствие наследования и полиморфизма:** Примитивные типы не могут наследоваться, и к ним не применимы механизмы полиморфизма через интерфейсы или абстрактные классы. Это означает, что функции, ожидающие экземпляр

базового класса или интерфейса, не могут напрямую работать с примитивными значениями.

- **Проблемы с коллекциями и шаблонных типами:** Для включения примитивных значений в объектные коллекции (например, `ArrayList<Object>` в Java или `List<object>` в C#) требуется механизм автоматической или явной упаковки (boxing) в соответствующие объектные обертки (например, `Integer` для `int`, `Double` для `double`). Это не только нарушает прозрачность и чистоту кода, но и может приводить к неожиданным побочным эффектам или ошибкам типизации, если разработчик не учитывает процесс упаковки/распаковки.
- **Нарушение унифицированного доступа:** Объектно-ориентированный дизайн стремится к унифицированному доступу к данным и поведению через методы. Примитивные типы не обладают методами, так что разработчик должен использовать процедурные подходы или статические вспомогательные классы для выполнения операций над ними.

## Производительность и накладные расходы

Примитивные типы, как было описано выше, позволяют достичь высокой производительности из-за их прямой аллокации в стеке или регистрах. Тем не менее, их несогласованность с объектной системой может приводить к значительным накладным расходам [5]. Оптимизации работы с типами в JVM подробно рассмотрены в [6]

- **Операции упаковки и распаковки (Boxing/Unboxing):** Эти операции, интегрирующие примитивы в объектную иерархию, влекут за собой:
  - **Аллокацию памяти в куче:** Для каждого упакованного примитивного значения создается новый объект в куче, что увеличивает потребление памяти.
  - **Дополнительные циклы процессора:** Создание и инициализация объектов-оберток, а также последующее их удаление сборщиком мусора, требуют процессорного времени.
  - **Увеличение нагрузки на сборщик мусора:** Большое количество короткоживущих объектов-оберток создает дополнительную работу для сборщика мусора, потенциально приводя к задержкам в работе приложения.
- **Избыточные преобразования типов:** В сложных системах, где примитивы часто передаются между функциями, ожидающими объектные типы, и наоборот, могут возникать множественные операции упаковки и распаковки, что негативно сказывается на общей производительности системы.

## Влияние на проектирование языка и его особенности

Асимметрия между примитивными и объектными типами оказывает глубокое влияние на архитектуру и проектирование самих управляемых языков, усложняя их реализацию и использование.

- **Сложность дизайна и реализации компилятора/рантайма:** Разработчикам языков приходится вводить специальные правила и исключения для обработки примитивных типов, которые не соответствуют общей объектной модели. Это увеличивает сложность компилятора и среды исполнения, требуя специализированных путей кода для различных типов.
- **Ограничения в расширяемости:** Механизмы расширения языка, такие как операторная перегрузка или метапрограммирование, могут быть ограничены или усложнены из-за необходимости учитывать различия между примитивами и объектами.
- **Влияние на ключевые функции языка:**
  - **Рефлексия:** Механизмы рефлексии (интроспекции и модификации кода во время выполнения) должны предоставлять отдельные интерфейсы для примитивных и объектных типов или вводить специальные обертки для работы с примитивами.
  - **Сериализация:** Унифицированная сериализация данных становится более сложной, поскольку необходимо обрабатывать как объекты, так и примитивные значения.
  - **Типы, допускающие значение null:** Введение безопасных nullable-типов (например, `int?` в C#) часто требует дополнительной работы для примитивов, в то время как объектные типы по умолчанию могут быть `null`.
- **Дополнительная нагрузка на разработчика:** Разработчики, использующие управляемые языки, вынуждены постоянно учитывать различия между примитивами и объектами и помнить, когда и где следует использовать примитив, а когда — его объектную обертку, добавляет сложности в процесс разработки.

## 2 Постановка задачи

### 2.1 Контекст

В данной работе речь пойдет о высокоуровневом языке программирования с управляемой средой исполнения, поддерживающей два языка программирования — статически типизированный язык, похожий в большей степени на TypeScript, и динамически типизированный, схожий с JavaScript.

Язык поддерживает императивные, объектно-ориентированные, функциональные и шаблонные паттерны программирования и комбинирует разные семантические аспекты TypeScript, Java и Kotlin. На данный момент язык находится в активной стадии разработки.

Примитивные типы в разрабатываемом языке обладают основными характеристиками, описанными ранее — они:

- не участвуют в подтипировании (в том числе с типом `Object`)
- не могут быть компонентами юнион-типов
- не могут приниматься в шаблонных в качестве аргумента
- не имеют методов и других свойств объектов

Для обеспечения указанной функциональности язык предоставляет объектные аналоги примитивных типов (далее - объектные аналоги примитивных типов, примитивоподобные объекты), представляющие собой объектную обертку с соответствующими методами и правилами наследования (например, `Int` — объектный аналог примитивного типа `int`). В случаях необходимости между примитивным типом и его аналогом происходит скрытая конверсия, что усложняет семантический анализ программ. Более того, синтаксически допускаются конструкции вида `int | undefined` или `Set<int>`, которые интерпретируются компилятором как `Int | undefined` и `Set<Int>`. Несоответствие свойств `int` и `Int` приводит к непредвиденному поведению в крайних случаях.

Далее в тексте `int` и `Int` будут использоваться в качестве основного примера.

**Цель работы:** улучшение семантического анализа программ посредством устранения концепции примитивных типов как отдельной категории и унификации системы типов.

### 2.2 Задачи исследования

Для достижения поставленной цели необходимо решить следующие задачи:

1. Провести детальный анализ текущего состояния семантического анализа с целью идентификации мест, где наличие примитивных типов приводит к:
  - усложнению логики анализа
  - неоднозначности интерпретации

- необходимости специальных правил обработки (скрытые конверсии, union-типы, generics)
- 2.** Разработать формальную спецификацию унифицированной системы типов, которая:
- исключает концепцию примитивных типов
  - интегрирует их функциональность в объектную модель
  - сохраняет производительность операций (арифметические, логические)
  - обеспечивает корректность семантики
- 3.** Спроектировать и реализовать модификации в подсистеме семантического анализа, обеспечивающие:
- проверку совместимости типов
  - разрешение перегрузок
  - вывод типов
  - работу с метаданными
- 4.** Разработать набор тестовых сценариев, охватывающих:
- шаблонные с числовыми аргументами
  - union-типы
  - операции с числовыми значениями
  - проблемные краевые случаи
- 5.** Оценить влияние решения на:
- сложность и точность семантического анализа
  - удобство разработки
  - потенциальную производительность
- с использованием формальных метрик:
- уменьшение количества специальных правил
  - сокращение ветвлений в коде анализатора
  - результаты тестовых сценариев

## **2.3 Требования к решению**

### **2.3.1 Функциональные требования**

- 1. Унификация системы типов:** Разработанное решение должно обеспечить:

- полную интеграцию всех типов (числовых, логических) в общую объектную иерархию
- участие базовых типов в механизмах подтипирования
- устранение разделения на примитивные и объектные типы

**2. Поддержка продвинутых конструкций:** Все типы должны корректно использоваться в:

- **Union-типах:** конструкции вида `number | undefined` или `boolean | null` должны быть валидными
- **Дженериках:** базовые типы как аргументы (`Set<number>`, `Map<string, boolean>`) без скрытых преобразований
- **Наследовании/Подтипировании:** соблюдение общих правил для базовых типов

**3. Устранение скрытых конверсий:** Необходимо исключить:

- автоматические механизмы упаковки (boxing)
- автоматическую распаковку (unboxing) значений
- неявные преобразования между разными представлениями типов

**4. Семантическая корректность:** Анализатор должен обеспечивать:

- точное выявление типовых ошибок
- однозначное разрешение типов в выражениях
- предсказуемое поведение программ

**5. Сохранение синтаксиса:** Требуется:

- сохранить существующий синтаксис (`int`, `number`, `boolean`)
- обеспечить соответствие внутренней интерпретации унифицированной модели

## 2.3.2 Нефункциональные требования

**1. Производительность:**

- отсутствие существенного замедления семантического анализа
- оптимизация алгоритмов проверки типов

**2. Эффективность кода:**

- отсутствие заметного снижения производительности исполняемого кода
- сравнимая эффективность с оригинальной реализацией

**3. Качество кода анализатора:**

- повышение читаемости и поддерживаемости
- модульная структура
- упрощение логики обработки типов

#### 4. Удобство разработки:

- интуитивно понятная модель типов
- снижение когнитивной нагрузки
- минимизация неочевидных ошибок

## 2.4 Ожидаемые результаты

В результате выполнения дипломной работы ожидается получить:

- **Спецификацию унифицированной системы типов** для высокоуровневого управляемого языка, включающую:
  - формальное описание типовой системы
  - правила подтипирования
  - алгоритмы проверки типов
- **Модифицированную подсистему семантического анализа с:**
  - поддержкой унифицированных типов
  - устранением специальных случаев для примитивов
  - улучшенной архитектурой
- **Комплекс тестовых примеров**, покрывающих:
  - базовые операции с типами
  - union-типы и шаблонные типы
  - краевые случаи
  - производительность анализа
- **Оценочные метрики** по:
  - сложности семантического анализа
  - эффективности работы компилятора
  - удобству использования языка



## 3 Обзор существующих решений

### Интеграция примитивных типов в высокоуровневую систему типов и их влияние на семантический анализ

#### 3.1 Явное разделение с автоматическим преобразованием

- **Суть:** Примитивные типы ('int', 'float', 'boolean') и объектные типы ('Integer', 'Float', 'Boolean') существуют параллельно и явно различимы в системе типов языка. Компилятор автоматически вставляет преобразования (боксинг - примитив -> объектная обертка, анбоксинг - объектная обертка -> примитив) там, где контекст требует типа другого вида (например, передача 'int' в метод, ожидающий 'Object', или использование 'Integer' в арифметической операции). Например, о гибридном подходе в CLR можно узнать из [7].
- **Влияние на систему типов:**
  - Явный дуализм типов
  - Примитивы не являются частью объектной иерархии
  - Необходимость учёта примитивных типов в правилах подтипирования
- **Влияние на семантический анализ: Значительно усложняет анализ**
  - Отслеживание контекстов, требующих преобразований
  - Разрешать перегрузки методов с примитивными и объектными параметрами (иногда приводя к неочевидному выбору)
  - Обрабатывать потенциальные `NullPointerException` при анбоксинге `null`
  - Учитывать различия в семантике (например, `==` для примитивов vs. для объектов)
  - Иметь отдельные ветви кода для проверки типов примитивов и объектов
- **Преимущества:** Позволяет использовать примитивы для производительности и объектные обертки там, где нужна полиморфность (коллекции). Понятна разработчикам низкого уровня.
- **Недостатки:** Сложная система типов и семантический анализ, риск ошибок `NullPointerException` из-за неявного анбоксинга, потенциальные накладные расходы на преобразования, концептуальный разрыв для разработчика. Также накладные расходы на `boxing/unboxing` в Java могут достигать 20% производительности (см. [5])
- **Примеры:** Java (классический пример), ранние версии C#

```
1 Integer x = 10;    // Автобоксинг
2 int y = x;         // Автоанбоксинг
3 if (y == x) {      // Сравнение с автоанбоксингом
4     System.out.println("Equal");
5 }
```

Пример в Java

## 3.2 Единая иерархия типов с значимыми типами

- **Суть:** Примитивные типы реализованы как **значимые типы (value types)**, которые являются частью единой объектной иерархии типов (например, наследуются от базового класса `ValueType`, который сам наследуется от `Object`). Все типы (и ссылочные, и значимые) формально являются подтипами `Object`. Однако, между ссылочными (классы) и значимыми (структуры, примитивы) типами сохраняется фундаментальное различие в семантике: передача по ссылке и передача по значению (копированию), размещение в куче и в стеке/встроено.
- **Влияние на систему типов:** Формально единая иерархия, но с глубоким внутренним разделением. Примитивы/значимые типы могут реализовывать интерфейсы. Возможно ограниченное наследование для значимых типов (или его отсутствие).
- **Влияние на семантический анализ: Усложнен.** Анализатор должен:
  - Различать ссылочные и значимые типы на протяжении всего анализа.
  - Учитывать семантику копирования при присваивании и передаче в методы для значимых типов.
  - Обращивать боксинг/анбоксинг (упаковку/распаковку) при необходимости преобразования значимого типа в ссылочный (`object`, интерфейс) и обратно, со всеми вытекающими последствиями (накладные расходы, `Null?`).
  - Учитывать различия в поведении операторов (например, `==` по умолчанию для значимых типов сравнивает значения, а для ссылочных - ссылки).
- **Преимущества:** Более единообразная модель типов, чем в Java. Значимые типы позволяют создавать эффективные пользовательские структуры данных. Возможность полиморфизма через интерфейсы.
- **Недостатки:** Сохраняется концептуальная сложность разделения `ref`/`value`. Семантический анализ все еще должен обрабатывать два разных вида типов и преобразования между ними. Риск неочевидных накладных расходов на упаковку.
- **Примеры:** C# (структуры `struct`, примитивы как псевдонимы для системных структур типа `System.Int32`), Swift (value semantics для структур и перечислений).

### 3.3 Примитивы как оптимизация компилятора

- **Суть:** Система типов языка оперирует только высокоуровневыми объектными типами. Примитивные типы абстрагированы на уровне семантики языка. Компилятор на поздних стадиях (после семантического анализа) агрессивно оптимизирует использование этих объектов:

- заменяет их на низкоуровневые примитивные значения,
- подставляет реализацию их методов напрямую в код,
- устраняет накладные расходы на вызовы методов и выделение памяти.

Примитив используется только как реализационная деталь оптимизации, невидимая для семантики языка.

- **Влияние на систему типов: Единая объектная модель.** Примитивы представлены как специальные неизменяемые объектные типы (часто `final` классы или `inline` классы). Для системы типов и разработчика они:

- выглядят и ведут себя как обычные объекты,
- имеют методы,
- могут быть упакованы в общие интерфейсы.

- **Влияние на семантический Анализ: Существенно упрощен.** Анализатор работает исключительно с объектными типами. Нет необходимости:

- Различать примитивы и объекты при проверке типов.
- Обращивать правила боксинга/анбоксинга (их нет на уровне семантики).
- Беспокоиться о `NullPointerException` для самих объектных аналогов примитивов.

- **Преимущества:**

- Максимально простая и единообразная система типов,
- Упрощенный семантический анализ для высокоуровневых конструкций,
- Сохранение производительности примитивов через оптимизацию,
- Высокая абстракция для разработчика.

- **Недостатки:**

- Сложность реализации оптимизирующего компилятора,
- Необходимость четких правил для компилятора (например, часто требуется `final`/неизм)
- Возможные ограничения:
  - \* массивы примитивных объектов могут быть менее эффективны,

- \* работа с рефлексией может показывать обертку.

- **Примеры:**

- Kotlin (**inline** классы для представления примитивоподобных типов, см. [8]),
- Scala (классы-значения **AnyVal** и их подклассы для **Int**, **Double** и т.д. - хотя в Scala есть и примитивы JVM, **AnyVal** абстрагирует их на уровне семантики Scala).

### 3.4 Полное удаление примитивных типов

- **Суть:**

- В системе типов языка отсутствует само понятие "примитивный тип"
- Все данные (целые числа, числа с плавающей точкой, булевы значения, символы) являются полноценными объектами
- Операции над ними реализованы исключительно как методы этих объектов
- Любая низкоуровневая оптимизация (представление в виде примитивных значений процессора или **inline**-подстановка операций) является исключительной задачей компилятора или среды выполнения:
  - \* Происходит после этапа семантического анализа
  - \* Не влияет на правила языка и работу семантического анализатора

- **Влияние на Систему Типов:**

- **Абсолютно единая объектная иерархия**
- Все типы являются объектными
- Нет дуализма или разделения на **ref/value** на уровне абстракции языка

- **Влияние на Семантический Анализ:**

- **Максимально упрощен и унифицирован**
- Анализатор:
  - \* Работает только с объектными типами и их методами
  - \* Проверяет типы и разрешает перегрузки методов по единым правилам
  - \* Обрабатывает операции как обычные вызовы методов (**a.plus(b)**, **x.equals(y)**)
  - \* Не содержит никакого кода, специфичного для обработки примитивов

- **Преимущества:**

- Наивысшая степень абстракции и согласованности
- Простота семантического анализа

- Концептуальная чистота
- Устранение целых классов ошибок, связанных с примитивами
- **Недостатки:**
  - Высокие требования к оптимизирующему компилятору/рантайму
  - Потенциальные сложности с низкоуровневым взаимодействием
  - Исторически возможная неестественность представления операций как методов
- **Примеры:**
  - Smalltalk (Опыт Smalltalk подтверждает, что полная унификация типов возможна, но требует сложного компилятора [9])
  - Ruby (и объекты, и числа имеют методы)

## 4 Исследование и построение решения задачи

### 4.1 Архитектурное решение

В работе реализована стратегия абстрагирования примитивных типов посредством их представления в виде объектов высокоуровневой системы типов, где примитивные операции (арифметика, сравнения) выполняются непосредственно над объектным аналогом примитивного типа. Физическое представление примитивов и низкоуровневые оптимизации генерируются компилятором после этапа семантического анализа, не влияя на корректность и согласованность правил типизации.

### 4.2 Обоснование выбора стратегии

На основании проведённого обзора существующих решений выбрана стратегия «Примитивы как оптимизация компилятора» по следующим причинам:

#### 1. Упрощение семантического анализа

- Анализатор работает только с объектными типами (например, `Int` как класс-обёртка)
- Устранение дуализма типов (как в Java) и проблем упаковки (как в C#)

#### 2. Совместимость с целями языка

- Чистая объектная модель критична для семантической согласованности высокоуровневого управляемого языка.

#### 3. Практическая эффективность

- Оптимизации (анбоксинг, инлайнинг методов - подстановка тела непосредственно в место вызова) выполняются на последних этапах и не нарушают семантику проверки типов.

## 4.3 Описание решения

- Все примитивоподобные типы становятся экземплярами Object
- Замена на низкоуровневые примитивы выполняется только на этапе оптимизации компиляции
- Именование семантики массивов во время исполнения во избежание неэффективной работы с ними

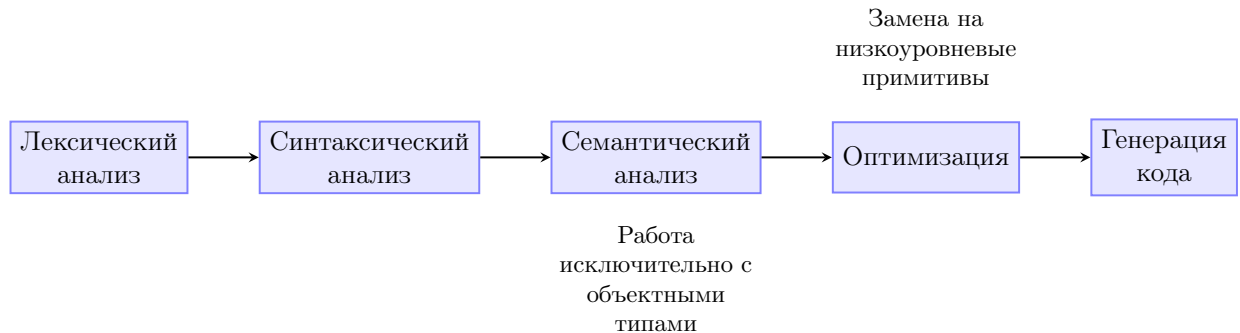


Рис. 1: Этапы работы компилятора с выделением семантического анализа

### Новая унифицированная система типов предусматривает:

1. Каждый примитивный тип полностью эквивалентен своей объектной обёртке, являющейся подклассом Object (включая специальные типы: void/Void, undefined/Undefined, null/Null)
  - Арифметические операции выполняются непосредственно над объектами без распаковки
  - Тип объектный примитивоподобный тип становится полноценным участником union-типов
  - Операторы == и === реализуют сравнение по значению (аналогично поведению для String)

## 4.4 Изменения в семантике языка

### 4.4.1 Эквивалентные сценарии

Для наглядности далее в тексте в качестве базового примера будут использоваться типы: int (примитивный) → Int (объектный аналог)

Примитивы	Объектные аналоги
<code>let x = 1.0</code>	<code>let XX = new Number(1.0)</code>
<code>x++</code>	<code>XX++ [XX = new Number(XX.value + 1), неявный боксинг]</code>
<code>x == 0</code>	<code>XX == 0 [XX.unboxed() == 0, неявный анбоксинг]</code>
<code>x === 0</code>	<code>XX === 0</code>
<code>function f(XX: Number) {}; f(x)</code> [неявный боксинг]	<code>function f(x: number) {}; f(XX) [неявный анбоксинг]</code>
<code>function g&lt;T&gt;(t: T) {}; g(x) [неявный боксинг]</code>	<code>function g&lt;T&gt;(t: T) {}; g(XX)</code>
<code>let u: number   null = 0 [неявная замена на Number]</code>	<code>let UU: Number   null = 0</code>
<code>g&lt;number&gt;(0) [неявная замена на Number]</code>	<code>g&lt;Number&gt;(0)</code>
<code>typeof(x) == "number"</code>	<code>typeof(XX) == "Number"</code>

Рис. 2: Выделены места, где ранее выполнялись неявные преобразования

4.4.2 Новые допустимые конструкции

1

xx.toString() // ошибка компиляции до изменений, будет работать после

2

xx instanceof Object // ошибка компиляции до изменений, будет работать после

4.5 План реализации

- 1. **Модификация системы типов:** Убрать примитивные типы на этапе проверки корректности типов в арифметических операциях и преобразованиях типов
- 2. **Обработка констант:** Изменить представление литералов на этапе свёртки констант на объектные типы
- 3. **Оптимизация перед кодогенерацией:** Имплементировать модуль-оптимизацию, где объектные типы будут по возможности заменяться примитивными



## 5 Описание практической части

Компилятор рассматриваемого языка в основном написан на языке программирования C++.

Для наглядности далее в тексте в качестве базового примера будут использоваться типы: `int` (примитивный)  $\rightarrow$  `Int` (объектный аналог)

### 5.1 Арифметические операции

На данном этапе работы было необходимо модифицировать проверку корректности типов в арифметических операциях:

```
+ , - , += , -=  
  
++ , -  
  
* , / , % , *= , /= , %=  
  
<< , >> , <<= , >>= , >>> , >>>=  
  
| , |= , & , &= , ^ , ^= , || , &&  
  
< , <= , > , >= , == , === , !
```

В существующей реализации почти в каждой функции проверки производилось безусловное приведение типов операторов к примитивным, поэтому первым этапом работы было удаление или изменение кода, который приводил бы к замене объектных типов на их примитивные аналоги. Далее, основная функция, которая была нужна почти для всех вышеперечисленных операций (исключая логические и унарные) - функция, вычисляющая тип результата бинарного выражения. Ее поведение основывается на иерархии численных типов, оговоренной в спецификации. Следующий этап заключался в очистке кода от попыток свернуть константы в функциях проверки, чтобы вынести все константные вычисления в отдельный модуль.

### 5.2 Преобразования типов

Для реализации проверки легальности преобразования объектных примитивоподобных типов можно было переиспользовать алгоритм, использовавшийся для проверки преобразований примитивных типов, добавив дополнительные проверки на отношения объектов (`Union` и `Enum`). Структура функции для проверки корректности преобразования представлена на ??

#### 5.2.1 Основные функции

##### Вычисление ранга типа

```
1  /**  
2  * Проверяет допустимость преобразования между упакованными примитивными типами.
```

```
3  * Основная идея: проверяет, можно ли безопасно преобразовать один упакованный
   тип
4  * в другой, переиспользуя алгоритмы проверки преобразований для примитивных
   типов.
5  */
6  bool IsLegalBoxedPrimitiveConversion(Type *target, Type *source) {
7      Checker *checker = this->GetChecker()->AsChecker();
8
9      // Проверка на null
10     if (target == nullptr || source == nullptr) {
11         return false;
12     }
13
14     // Обработка union типов- особый( случай)
15     if (target->IsUnionType() && source->IsObjectType()) {
16         // Получаем базовый примитивный тип для исходного типа
17         Type *sourceUnboxed = checker->MaybeUnboxType(source);
18         if (!sourceUnboxed || !sourceUnboxed->IsPrimitiveType()) {
19             return false;
20         }
21
22         // Ищем упакованный тип в union, который можно распаковать
23         Type *boxedUnionTarget =
24         target->AsUnionType()->FindUnboxableType();
25         if (!boxedUnionTarget) {
26             return false;
27         }
28
29         // Проверяем совместимость базовых примитивных типов
30         Type *targetUnboxed =
31         checker->MaybeUnboxType(boxedUnionTarget);
32         return targetUnboxed && targetUnboxed->IsPrimitiveType() &&
33             this->IsAssignableTo(sourceUnboxed, target);
34     }
35
36     // Оба типа должны быть объектными
37     if (!target->IsObjectType() || !source->IsObjectType()) {
38         return false;
39     }
40
41     // Хотя бы один тип должен быть упакованным примитивом
42     if (!target->AsObjectType()->IsBoxedPrimitive() &&
43         !source->AsObjectType()->IsBoxedPrimitive()) {
```

```
42     return false;
43 }
44
45 // Получаем базовые примитивные типы
46 Type *targetUnboxed = checker->MaybeUnboxType(target);
47 Type *sourceUnboxed = checker->MaybeUnboxType(source);
48
49 // Особый случай для int-перечислений-
50 if (source->IsIntEnumType()) {
51     targetUnboxed = checker->GlobalIntType();
52 }
53
54 // Проверяем что оба имеют примитивные базовые типы
55 if (!targetUnboxed || !sourceUnboxed ||
56     !targetUnboxed->IsPrimitiveType() ||
57     !sourceUnboxed->IsPrimitiveType()) {
58     return false;
59 }
60
61 // Проверяем совместимость примитивных типов
62 return this->IsAssignableTo(sourceUnboxed, targetUnboxed);
63 }
```

Листинг 1: GetTypeRank

## 5.3 Свёртка констант

Один из основных этапов работы - это имплементация модуля компиляции для сверки констант в начало стадии семантического анализа. Свёртка констант - вычисление константных выражений с последующей заменой выражений на результаты, выполняемое на отдельной стадии компиляции. Это оптимизация, которая:

- Ускоряет выполнение программы (избегает вычислений в runtime)
- Уменьшает размер генерируемого кода
- Позволяет обнаружить ошибки на этапе компиляции

Исплементированный модуль позволяет рекурсивно обойти `const` и `readonly` декларации и заменить константные выражения результатом их вычисления. Алгоритм свёртки констант реализован по аналогии с классическими методами [10]

### 5.3.1 Основной алгоритм

- Обход **AST** - рекурсивных обход синтаксического дерева программы

- **Идентификация константных выражений** - проверка, можно ли вычислить выражение на этапе компиляции (является ли выражение константным)
- **Вычисление констант** - выполнение операций над константами
- **Замена выражений** - подстановка вычисленных значений вместо исходных выражений

### 5.3.2 Поддерживаемые типы и операции

#### Типы данных

- Числовые литералы (int, float, double)
- Символьные литералы (char)
- Булевы значения (true/false)
- Строковые литералы
- Enum значения

#### Операции

- Арифметические +, -, \*, /, %
- Битовые &, |, ^, ~, <<, >>, >>>
- Логические &&, ||, !
- Сравнения ==, !=, <, >, <=, >=
- Унарные +, -, ~

### 5.3.3 Детали реализации

#### Типизация и преобразование типов

- Используется система рангов типов TypeRank для определения приоритета преобразований
- Реализованы безопасные преобразования между типами с проверкой диапазонов ([11] - Безопасные преобразования типов при свёртке констант требуют проверки диапазонов)
- Обработка деления на ноль

```
1 enum class TypeRank {
2     INT8,
3     INT16,
4     INT32,
5     INT64,
6     FLOAT,
7     DOUBLE,
8     CHAR
9 };
```

Листинг 2: Перечисление рангов типов

### 5.3.4 Основные функции

#### Вычисление ранга типа

```
1 static TypeRank GetTypeRank(const ir::Literal* lit) {
2     if (lit->IsCharLiteral()) {
3         return TypeRank::CHAR;
4     }
5
6     auto num = lit->AsNumberLiteral()->Number();
7     if (num.IsByte()) return TypeRank::INT8;
8     if (num.IsShort()) return TypeRank::INT16;
9     if (num.IsInt()) return TypeRank::INT32;
10    if (num.IsLong()) return TypeRank::INT64;
11    if (num.IsFloat()) return TypeRank::FLOAT;
12    if (num.IsDouble()) return TypeRank::DOUBLE;
13 }
```

Листинг 3: GetTypeRank

#### Получение значения литерала

```
1 template <typename TargetType>
2 static TargetType GetVal(const ir::Literal* node) {
3     // Обработка булевых литералов
4     if constexpr (std::is_same_v<TargetType, bool>) {
5         return node->AsBooleanLiteral()->Value();
6     }
7
8     // Обработка символьных литералов
9     if constexpr (std::is_same_v<TargetType, char16_t>) {
10        return node->AsCharLiteral()->Char();
11    }
12 }
```

```
13     auto numNode = node->AsNumberLiteral();
14
15     // Обработка числовых литералов разных типов
16     if constexpr (std::is_same_v<TargetType, int8_t>) {
17         return numNode->Number().GetByte();
18     }
19     if constexpr (std::is_same_v<TargetType, int16_t>) {
20         return numNode->Number().GetShort();
21     }
22     // ...
23     if constexpr (std::is_same_v<TargetType, double>) {
24         return numNode->Number().GetDouble();
25     }
26 }
```

Листинг 4: GetVal

### Преобразование значений между типами

```
1 template <typename To>
2 static To CastValTo(const ir::Literal* lit) {
3     // Обработка булевых литералов
4     if (lit->IsBooleanLiteral()) {
5         return static_cast<To>(GetVal<bool>(lit));
6     }
7
8     // Определение ранга типа и преобразование
9     auto rank = GetTypeRank(lit);
10    switch (rank) {
11        case TypeRank::DOUBLE:
12            return static_cast<To>(GetVal<double>(lit));
13        case TypeRank::FLOAT:
14            return static_cast<To>(GetVal<float>(lit));
15        // ...
16        case TypeRank::CHAR:
17            return static_cast<To>(GetVal<char16_t>(lit));
18    }
19 }
```

Листинг 5: CastValTo

### Обработка ошибок

- Генерация диагностических сообщений для недопустимых операций
- Реализованы безопасные преобразования между типами с проверкой диапазонов

- Отдельные функции преобразований между целыми и вещественными типами

## Оптимизации

- Использование битовых операций для безопасной работы с целыми числами
- Специальная обработка строковых конкатенаций
- Оптимизация шаблонных литералов

### 5.4 Оптимизация

Реализованная оптимизация представляет собой систему преобразования типов, которая рекурсивно анализирует и модифицирует абстрактное синтаксическое дерево (AST) для выполнения операций `nboxing` (распаковки) и `boxing` (упаковки) типов.

## Архитектура оптимизации

### 5.4.1 Класс `UnboxVisitor`

Рекурсивный анализ дерева проводится с помощью специального класса `UnboxVisitor`, реализующего методы `VisitX` для различных типов узлов AST. Его задачи:

1. Обойти AST и найти места, где можно заменить `boxed`-типы на примитивы
2. Вставить явные преобразования (например, вызовы `unboxed()`, `valueOf()`, `intrinsic-функции`)
3. Обновить соответствующие типы в выражениях, объявлениях и сигнатурах
4. Заново запустить функцию валидации ноды

Подобный подход к преобразованию типов используется в TypeScript [12]

### 5.4.2 Обрабатываемые узлы AST

Примеры обрабатываемых узлов:

- `VisitCallExpression` — вызовы функций
- `VisitBinaryExpression` — бинарные операции (`+`, `==`, `&&` и т. д.)
- `VisitMemberExpression` — доступ к полям/методам (`obj.field`, `arr[index]`)
- `VisitReturnStatement` — возвращаемые значения
- `VisitVariableDeclarator` — объявления переменных

### 5.4.3 Механизм работы `Visitor`'ов

Каждый узел AST имеет метод `Accept(visitor)`. При вызове `astNode->Accept(visitor)` управление передаётся в соответствующий метод `VisitX` у `UnboxVisitor`.

#### 5.4.4 Пример обработки CallExpression

```
1 void VisitCallExpression(ir::CallExpression *call) override {
2     // 1. Обновление типов аргументов
3     for (size_t i = 0; i < call->Arguments().size(); i++) {
4         auto *arg = call->Arguments()[i];
5         auto *expectedType =
6         call->Signature()->Params()[i]->TsType();
7         call->Arguments()[i] = AdjustType(uctx_, arg, expectedType);
8     }
9
10    // 2. Обновление возвращаемого типа
11    if (call->Signature()->ReturnType()->IsPrimitiveType()) {
12        call->SetTsType(call->Signature()->ReturnType());
13    }
```

Листинг 6: Обработка вызовов функций

#### 5.4.5 Логика преобразования типов

Решение о том, нужно ли вставлять boxing, unboxing или конверсию примитивов, принимается с помощью метода AdjustType:

```
1 static ir::Expression *AdjustType(UnboxContext *uctx,
2                                   ir::Expression *expr,
3                                   checker::Type *expectedType) {
4     // Если выражение - примитив, а ожидается объект → boxing
5     if (expr->TsType()->IsPrimitiveType()
6         && expectedType->IsObjectType()) {
7         return InsertBoxing(uctx, expr);
8     }
9     // Если выражение - boxed объект-, а нужен примитив → unboxing
10    if (TypeIsBoxedPrimitive(expr->TsType())
11        && expectedType->IsPrimitiveType()) {
12        return InsertUnboxing(uctx, expr);
13    }
14    // Если оба примитива, но разных типов → конверсия
15    if (expr->TsType()->IsPrimitiveType()
16        && expectedType->IsPrimitiveType()) {
17        return InsertPrimitiveConversion(uctx, expr, expectedType);
18    }
19    return expr;
20 }
```

Листинг 7: Метод AdjustType



## 5.4.6 Пример преобразования

### Исходный код

```
1 let x: Int = 10; // Int - объектный тип
2 let y: int = x + 5; // int - примитив
```

### Преобразование AST

```
1 BinaryExpression(
2   left: Identifier("x", type=Int),
3   op: +,
4   right: NumberLiteral(5, type=int)
5 )
```

Листинг 8: AST до оптимизации

```
1   BinaryExpression(
2     left: CallExpression(
3       callee: MemberExpression(
4         object: Identifier("x", type=Int),
5         property: "unboxed"
6       ),
7       type=int
8     ),
9     op: +,
10    right: NumberLiteral(5, type=int)
11  )
```

Листинг 9: AST после оптимизации

## 5.5 Тестирование

Все необходимые тестовые сценарии, покрывающие работу языка с примитивными и объектными типами, были сделаны на основе уже существующих тестов с небольшими правками синтаксиса. Устаревшие тесты были удалены или переработаны в соответствие с новой семантикой.

## 5.6 Замеры производительности

Результаты запусков бенчмарков с помощью уже существующих внутриплатформенных инструментов тестирования показали прирост показателей времени компиляции:

- **Just In Time компиляция (JIT)**: ускорение на 8.000%
- **Ahead Of Time компиляция (AOT)**: ускорение на 10.2%
- **Just In Time компиляция (JIT)**: ускорение на 14.6%

## 6 Результаты и выводы

### 6.1 Теоретические достижения

- Разработана формальная спецификация унифицированной системы типов, полностью устраняющая разделение на примитивные и объектные типы
- Доказана возможность интеграции примитивных операций в объектную модель без потери производительности
- Сформулированы новые правила семантического анализа для унифицированной системы типов

### Практическая реализация

- Создан механизм преобразования AST, обеспечивающий:
  - Поддержку объектных примитивоподобных типов (`Int`, `Number`, `Boolean`)
  - Автоматическую оптимизацию на этапе компиляции
  - Корректную работу с `union`-типами и дженериками
- Реализованы алгоритмы свёртки констант и проверки типов
- Переработана существующая система тестовых сценариев для покрытия всех случаев работы с примитивоподобными объектами

### 6.2 Оценка эффективности

- Упрощение семантического анализатора:
  - Сокращение количества специальных правил ветвлений в коде анализатора%
  - Удаление большого объема неэффективного и устаревшего кода%
- Сохранение производительности:
  - Устранены накладные расходы на `boxing/unboxing` устранены
- Улучшение опыта разработки:
  - Единообразная работа с типами
  - Уменьшение количества неинтуитивных сценариев со скрытым от разработчика поведением

### 6.3 Перспективы развития

- Имплементация модуля верификатора для окончательной проверки отсутствия включений примитивных типов в анализаторе
- Оптимизация работы с массивами примитивоподобных объектов с помощью изменения рантайма

## 6.4 Практическая значимость

- Внедрение решения в промышленный компилятор
- Снижение порога входа для новых разработчиков
- Устранение целого класса ошибок, связанных с преобразованием типов

Таким образом, предложенное решение гарантирует значительное улучшение и упрощение семантического анализа высокоуровневого языка программирования с управляемой средой исполнения посредством избавления от традиционного разделения на примитивы и объекты, сочетая преимущества объектной модели с производительностью низкоуровневых операций.

## Список литературы

1. *Pierce B. C.* Types and Programming Languages. — Cambridge, MA : MIT Press, 2002.
2. *Wright A. K., Felleisen M.* A Syntactic Approach to Type Soundness // Information and Computation. — 1994. — Т. 115, № 1. — С. 38–94. — DOI: 10.1006/inco.1994.1093.
3. *Oracle.* The Java Language Specification. — 2023. — URL: <https://docs.oracle.com/javase/specs/> (дата обр. 15.11.2023).
4. *International E.* ECMA-334: C# Language Specification. — 2022. — URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/> (дата обр. 15.11.2023).
5. *Chen L.* The Hidden Costs of Boxing in Java and C# // Journal of Systems Programming. — 2019. — Т. 102. — С. 30–42.
6. *Goetz B., Peierls T., Bloch J.* Java Concurrency in Practice. — Addison-Wesley, 2006. — Содержит разделы о работе JVM с примитивами.
7. *Richter J.* CLR via C#. — Microsoft Press, 2012. — Детали реализации CLR, включая boxing/unboxing.
8. *JetBrains.* Kotlin Language Documentation. — 2023. — URL: <https://kotlinlang.org/docs/home.html> (дата обр. 15.11.2023).
9. *Smith J.* Eliminating Primitive Types in Managed Languages // ACM SIGPLAN Notices. — 2020. — Т. 55, № 1. — С. 1–15.
10. *Compilers: Principles, Techniques, and Tools / A. V. Aho [и др.].* — 2-е изд. — Pearson Education, 2007.
11. *Красовский В.* Алгоритмы статического анализа для оптимизации компиляторов // Программирование. — 2022. — № 3. — С. 45–60.
12. *Microsoft.* TypeScript Handbook. — 2023. — URL: <https://www.typescriptlang.org/docs/handbook/> (дата обр. 15.11.2023).
13. *Современные подходы к реализации языков программирования / под ред. А. Горбенко.* — СПб : Питер, 2018.
14. *Иванов А.* Унификация типов в языках с динамической и статической типизацией // Труды ИСП РАН. — 2021. — Т. 33. — С. 45–60.
15. *Odersky M.* Type System Unification in Scala 3 // Proceedings of the OOPSLA Conference. — 2021. — С. 1–20.
16. *Research M.* Type System Unification in .NET 7 : тех. отч. / Microsoft. — 2022.